4장. 변수와 함수

4.1. 변수 선언 및 초기화

4.1.1. 변수 선언법

- val(혹은 var) 변수명 : 타입 = 값
- val (value)는 Assign-once 변수, var (variable)은 Mutable 변수
- 타입 추론 지원

```
val data1: Int = 10
val data2 = 20
var data3 = 30

fun main(args: Array<String>) {
//    data2 = 40//error
    data3=40//ok~~~~
}
```

4.1. 변수 선언 및 초기화

4.1.2. 변수 초기화

- 변수 선언은 최상위(클래스 외부), 클래스 내부, 함수 내부에 선언.
- 최상위 레벨이나 클래스의 멤버 변수는 선언과 동시에 초기화해주어야 한다.
- 함수 내부의 지역 변수는 선언과 동시에 초기화하지 않더라도된다. 초기화한후 사용할 수 있다.

```
val topData1: Int//error
var topData2: Int//error

class User {
    val objData1: String//error
    var objData2: String//error

    fun some(){
        val localData1: Int//ok...
        var localData2: String//ok...

        println(localData1)//error

        localData2="hello"//ok...
        println(localData2)//ok...
    }
}
```

4.1. 변수 선언 및 초기화

4.1.3. null이 될 수 있는 변수와 null

- 코틀린에서는 null을 대입할 수 없는 변수와 있는 변수로 구분
- 변수에 null 값을 대입하려면 타입에 ? 기호를 이용하여 명시적으로 null이 될 수 있는 변수로 선언.

```
//val nonNullData: String = null//error
val nullableData1: String? = null
var nullableData2: String? = null

fun main(args: Array<String>) {
//    nonNullData="hello"//error
    nullableData2="hello"//ok
}
```

4.1. 변수 선언 및 초기화

4.1.4. 상수변수 선언

- 코틀린에서 변수는 프로퍼티(property)이다.
- val로 선언한 변수의 초깃값을 변경할 수는 없지만, 일반적인 상수변수와는 차이가 있다.
- const라는 예약어를 이용해 상수 변수를 만든다.
- 최상위 레벨로 선언할 때만 const 예약어를 사용 가능

```
const val myConst: Int = 10

//const var myConst2: Int = 10//error

class MyClass {
//    const val myConst3 = 30//error
}
fun some(){
//    const val myConst4= 40//error
}
```

4.2. 함수 사용법

4.1.2. 함수 선언

- fun 함수명(매개변수명 : 타입) : 리턴타입 { }

```
fun sum(a: Int, b: Int): Int {
    return a + b
}
```

- 매개변수 에는 var, val을 선언할 수 없다. 매개변수는 기본으로 val이 적용
- 의미있는 반환값이 없을 때는 Unit으로 명시
- Unit은 생략할 수 있으며 함수의 반환 타입이 선언되지 않았다면 기본으로 Unit이 적용

```
fun sum(a: Int, b: Int): Unit {
    //.....
}
```

```
fun sum(a: Int, b: Int) {
    //.....
}
```

4.2. 함수 사용법

- 함수내에 함수선언 가능

```
fun sum(a: Int, b: Int): Int {
    var sum=0
    fun calSum(){
        for(i in a..b){
            sum += i
        }
    }
    calSum()
    return sum
}
```

- Single expression function

```
fun some(a: Int, b: Int): Int {
    return a + b
}
```

```
fun some(a: Int, b: Int): Int = a  + b
```

```
fun some(a: Int, b: Int) = a  + b
```

4.2. 함수 사용법

4.2.2. 함수 오버로딩

```
fun some(a: String){
    println("some(a: String) call....")
}
fun some(a: Int){
    println("some(a: Int) call....")
}
fun some(a: Int, b: String){
    println("some(a: Int, b: String) call....")
}
```

4.2. 함수 사용법

4.2.3. 기본 인수와 명명된 인수

- default argument

```
fun sayHello(name: String){
    println("Hello!!"+name)
}
```

```
fun sayHello(name: String?){
    if(name==null){
        println("Hello!! kkang")
    }else {
        println("Hello!!"+name)
    }
}
```

```
fun sayHello(name: String = "kkang"){
    println("Hello!!"+name)
}
```

4.2. 함수 사용법

- named argument

```
fun sayHello(name: String = "kkang", no: Int){
    println("Hello!!"+name)
}
fun main(args: Array<String>) {
//    sayHello(10)//error
    sayHello("lee", 20)
    sayHello(no=10)
    sayHello(name="kim", no=10)
}
```

4.2. 함수 사용법

4.2.4. 중위표현식

- infix(중위 표현식) 이란 연산자를 피 연산자의 중간에 위치시킨 다는 개념
- 중위 표현식을 함수 호출에도 사용이 가능

```
infix fun Int.myFun(x: Int): Int {
    return x * x
}
class FunClass {
    infix fun infixFun(a: Int){
        println("infixFun call....")
    }
}
fun main(args: Array<String>) {
    val obj=FunClass()
    obj.infixFun(10)
    //중위 표현식
   obj infixFun 10

    println(10  myFun 10)
    println(10.myFun(10))
}
```

- 클래스의 맴버 함수로 선언되거나 혹은 클래스의 extension 함수인 경우
- 하나의 매개변수를 가지는 함수의 경우

4.2. 함수 사용법

4.2.5. 가변인수

```
fun <T> varargsFun(a1: Int, vararg array: T){
    for( a in array){
        println(a)
    }
}

fun main(args: Array<String>) {
    varargsFun(10, "hello", "world")
    varargsFun(10, 20, false)
}
```

4.2. 함수 사용법

4.2.6. 재귀함수

- 재귀함수란 함수 내에서 자신의 함수를 다시 호출하는 것

```
fun loopPrint(no: Int = 1){
    var count=1
    while(true){
        println("loopPrint..")
        if(no == count) return
        else count++
    }
}
```

```
fun recPrint(no: Int = 1, count: Int = 1){
    println("recPrint...")
    return if(no==count) return else recPrint(no -1, count)
}
```

4.2. 함수 사용법

- tailrec 라는 예약어를 이용해 재귀함수를 정의하게 하여 조금더 효율적인 재귀 함수를 만들수 있다.

```
tailrec fun tailrecPrint(no: Int = 1, count: Int = 1){
    println("tailrecPrint...")
    return if(no==count) return else tailrecPrint(no -1, count)
}
```

```
public static final void recPrint(int no, int count) {
   String var2 = "recPrint...";
   System.out.println(var2);
   if (no != count) {
      recPrint(no - 1, count);
   }
}

public static final void tailrecPrint(int no, int count) {
   while(true) {
      String var2 = "tailrecPrint...";
      System.out.println(var2);
      if (no == count) {
         return;
      }

      --no;
   }
}
```

4.2. 함수 사용법

- tailrec 는 꼬리 재귀함수의 경우만 추가 가능

```
tailrec fun sum(n: Int): Int {
    if (n <= 0) return n
    else return n + sum(n - 1)
}
tailrec fun sum2(n: Int, result: Int = 0): Int {
    if (n <= 0) return result
    else return sum2(n - 1, n + result)
}
```

```
public static final int sum(int n) {
   return n <= 0 ? n : n + sum(n - 1);
}

public static final int sum2(int n, int result) {
   while(n > 0) {
      int var10000 = n - 1;
      result += n;
      n = var10000;
   }

   return result;
}
```